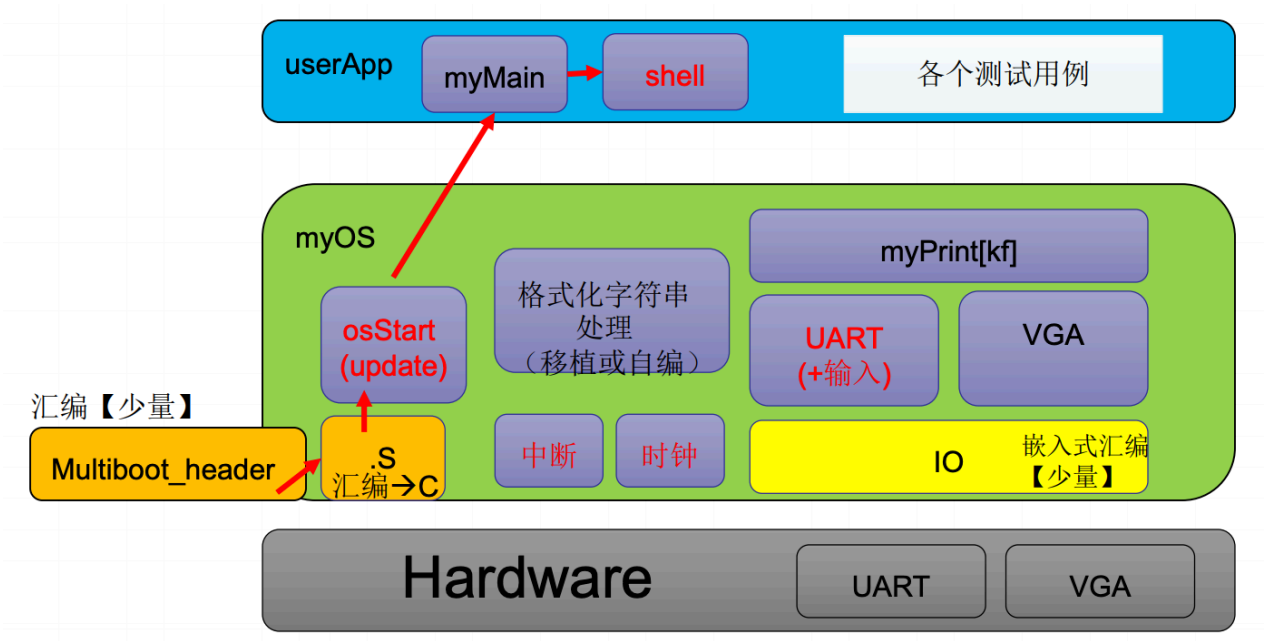
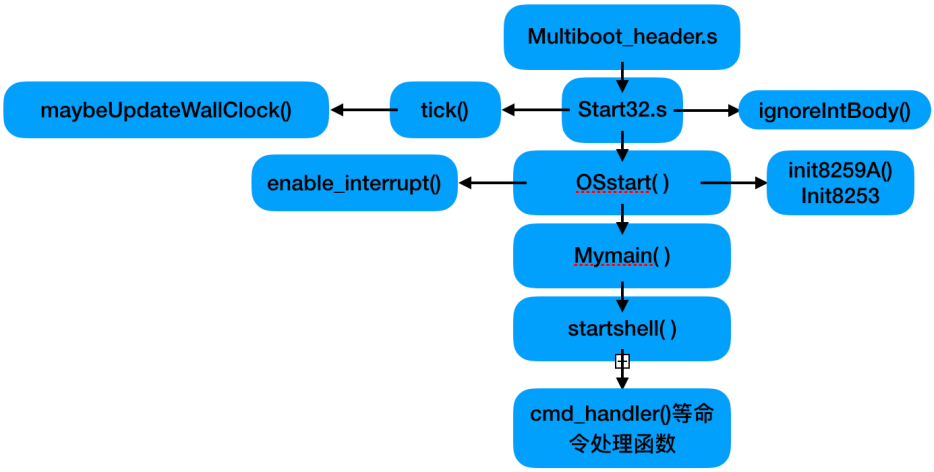
实验三 shell&Interrupt

软件框架及概述



概述：从Multiboot_header进入入操作系统内核(myOS)，然后开始为进入入C程序准备好上下文，初始化操作系统，设置好中断处理和时钟。再调用用userApp入口myMain。在myMain中测试功能，包括通过调用Myprint[tf]函数实现 VGA输出和串口输出以及用户的shell功能。

主流程及其实现



主流程图

主流程说明：multiboot_header.s调用_start入口，myOS中的Start32.s初始化了一个中断描述符表IDT和寄存器IDTR，将所有中断处理程序初始化为一个缺省处理函数ignore_int1()，设置时钟中断(包括后续的tick维护、墙钟维护和显示)，提供了osStart()入口，做好第一次调用C语言入口前的准备，进入osStart()后，先初始化操作系统，包括初始化i8259、i8253，并开中断，再调用与userApp之间的接口myMain，进入myMain后，调用startshell()进入自定义shell，显示交互界面，接受和处理命令行。

主要功能模块及其实现（含源代码说明）

模块一 基本功能

1. init8259A()

```
extern void outb (unsigned short int port_to, unsigned char value);
extern unsigned char inb(unsigned short int port_from);
void init8259A(void)        //初始化 PIC i8259
{
    outb(0x21,0xff);
    outb(0xA1,0xff);

    outb(0x20,0x11);
    outb(0x21,0x20);
    outb(0x21,0x04);
    outb(0x21,0x03);

    outb(0xA0,0x11);
    outb(0xA1,0x28);
    outb(0xA1,0x02);
    outb(0xA1,0x01);

}
```

2. Init8253()

```
extern void outb (unsigned short int port_to, unsigned char value);
extern unsigned char inb(unsigned short int port_from);
void init8253(void)         //初始化PTI: i8253, 时钟中断的频率定为100HZ, 分频参数为11932
{
    outb(0x43,0x34);
    outb(0x40,11932 & 0xff);
    outb(0x40,11932 >> 8);
    outb(0x21,inb(0x21)&0xFE);
}
```

3. tick()

流程：tick() ------> maybeUpdateWallClock()

```
extern void maybeUpdateWallClock(void);
int count=0;     //维护tick

void tick(void)          //处理周期性时钟中断
{
    maybeUpdateWallClock();
    ++count;
}
```

4. 维护时钟和显示时钟，

   流程： maybeUpdateWallClock()------>setWallClock()------>setWallClockHook()------>displayClock()------> getWallClock()

```
int hh=0, mm=0, ss=0, ms=0;
extern int count;

void setWallClock(int h, int m, int s)       //初始化墙钟
{
    hh=h;
    mm=m;
    ss=s;
}

void getWallClock(int *h, int *m, int *s)        //读取墙钟
{
        *h = hh;
        *m = mm;
        *s = ss;
}

void displayClock()                          //右下角显示墙钟
{
    int hour,minute,second;
    getWallClock(&hour,&minute,&second);

    int where = 0xb8000 + 25*80*2;
    *(unsigned char *)(where - 2) = (second % 10) + '0';
    *(unsigned char *)(where - 1) = 0x6;
    *(unsigned char *)(where - 4) = (second - second % 10) / 10 + '0';
    *(unsigned char *)(where - 3) = 0x6;

    *(unsigned char *)(where - 6) = ':';
    *(unsigned char *)(where - 5) = 0x6;

    *(unsigned char *)(where - 8) = (minute % 10) + '0';
    *(unsigned char *)(where - 7) = 0x6;
    *(unsigned char *)(where - 10) = (minute - minute % 10) / 10 + '0';
    *(unsigned char *)(where - 9) = 0x6;
```

```
36
37      *(unsigned char *)(where - 12) = ':';
38      *(unsigned char *)(where - 11) = 0x6;
39
40      *(unsigned char *)(where - 14) = (hour % 10) + '0';
41      *(unsigned char *)(where - 13) = 0x6;
42      *(unsigned char *)(where - 16) = (hour - hour % 10) / 10 + '0';
43      *(unsigned char *)(where - 15) = 0x6;
44
45  }
46
47  void setWallClockHook(void (*func)(void))           //设置hook
48  {
49      (*func)();
50  }
51
52  void maybeUpdateWallClock(void)      //根据tick维护墙钟
53  {
54      static int flag = 1;
55      if(1==flag)
56      {
57       setWallClock(12,26,38);
58       flag = 0;
59      }
60
61      ms+=10;                        //维护ms, ss, mm, hh
62      if(ms>=1000) {ms=0;ss++;}
63      if(ss>=60) {ss=0;mm++;}
64      if(mm>=60) {mm=0;hh++;}
65      if(hh>=24) hh=0;
66
67      if(0==count%100) setWallClockHook(displayClock);
68
69      void (*wallClock_hook)()=0;
70      if (wallClock_hook) wallClock_hook();
71  }
```

模块二 中断处理

1. 中断缺省处理程序 ignoreIntBody()

```
1  void ignoreIntBody(void)            //处理除时钟中断之外的所有其他中断
2  {
3      unsigned char *ptr = (unsigned char *)0xb8000;
4      char *str="Unknown interrupt1\0";
5      char c;
6      int row=24,col=0;
7      c = *str;
8      while (c!='\0'){
```

```
        ptr[(row * 80 + col) * 2] = c;
        ptr[(row * 80 + col) * 2 + 1] = 0x4;
        c = *(++str);
        col ++;
    }
}
```

2. 开中断，关中断

```
.text
.code32
    .globl enable_interrupt
    .globl disable_interrupt

enable_interrupt:       #开中断
    sti
    ret

disable_interrupt:      #关中断
    cli
    ret
```

模块三 shell

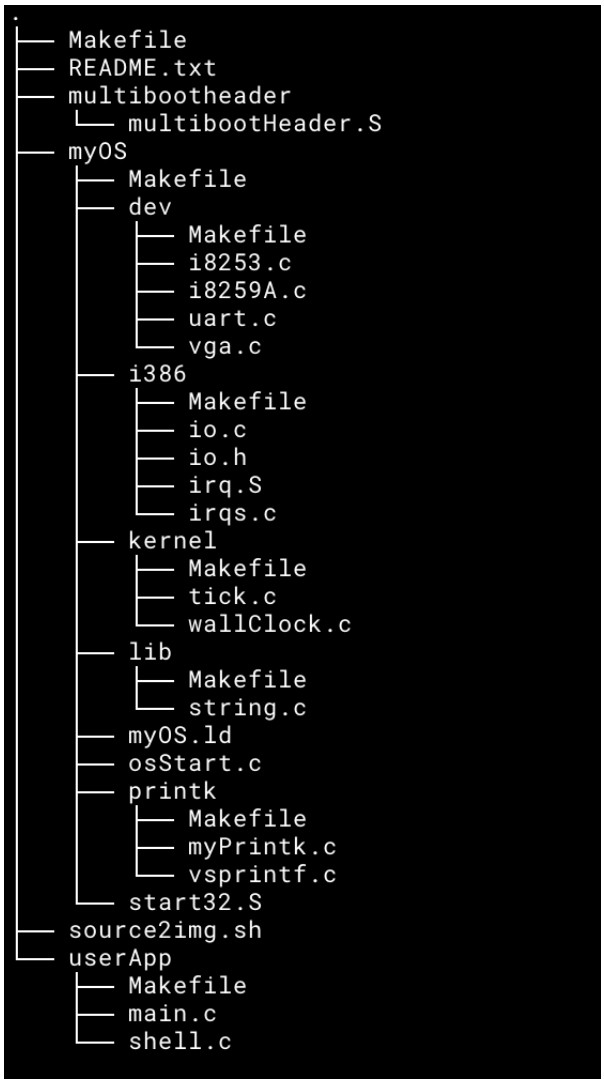
流程：startShell()------>cmd_handler()/help_handler()

```
#define NULL 0
extern unsigned char uart_get_char(void);
extern int myPrintk(int color,const char *format, ...);
extern int strcmp(char *str1, char *str2);
extern char *strtok(char *str, const char *delim);

// 命令处理函数
int cmd_handler(int, char **);
int help_handler(int, char **);

// 帮助处理函数
void help_help(void);

struct command {
    char *cmd;
    int (*func)(int argc, char *argv[]);
    void (*help_func)(void);
    char *desc;
} cmds[] = {
    {"cmd", cmd_handler, NULL, "list all commands"},
    {"help", help_handler, help_help, "help [cmd]"},
    {"", NULL, NULL, ""}
};
```

```
24
25
26  // help 的帮助
27  void help_help(void)
28  {
29      myPrintk(0x7,"\nUSAGE : help [cmd]\n");
30  }
31
32  // help 命令处理函数
33  int help_handler(int argc, char *argv[])
34  {
35      help_help();
36      if(argc>2) return 1;
37      if(argc==2){
38          int i=0;
39          while(cmds[i].func!=NULL){
40              if(strcmp(argv[1], cmds[i].cmd)==0) {
41                  myPrintk(0x7,"\n%s\n",cmds[i].desc);
42                  break;
43              }
44              ++i;
45          }
46          if(cmds[i].func==NULL)  return 1;
47      }
48      return 0;
49  }
50
51  // cmd 命令处理函数
52  int cmd_handler(int argc, char **argv)
53  {
54      if(argc!=1) return 1;
55      myPrintk(0x7,"\nlist all registed commands:\n");
56      myPrintk(0x7,"command name : description\n");
57      int i=0;
58      while(cmds[i].func!=NULL){
59          myPrintk(0x7,"%13s: %s\n",cmds[i].cmd,cmds[i].desc);
60          ++i;
61      }
62      return 0;
63  }
64
65
66  void startShell(void)
67  {
68      // TODO
69      int pos,i,argc;
70      unsigned char single[2];
71      unsigned char buf[200];
72      const char split[2]=" ";
```

```
    char *args[50];
    unsigned char c;
    while(1)
    {
        myPrintk(0x3,"rbzhang >:");
        pos=0;
        while((c=uart_get_char())!='\r')        //读入一行输入到buf[]
        {
            single[0]=c;
            single[1]='\0';
            myPrintk(0x7,single);
            buf[pos++]=c;
        }
        buf[pos]='\0';

        args[0]=strtok(buf,split);
        i=0;


        while(args[i]!=NULL){
            ++i;
            args[i]=strtok(NULL,split);     //分解一行输入，用指针数组args[]指向每个部分
        }


        if (args[0] == NULL) {
            myPrintk(0x7, "\n");
            continue;
        }

        argc=i;
        i=0;
        while(cmds[i].func!=NULL){              //选择待执行的内置命令
            if (strcmp(args[0],cmds[i].cmd) == 0) {
            if(1==(*cmds[i].func)(argc,args))myPrintk(0x7, "\nError arguments!\n");
            break;
            }
            i++;
        }

        if(cmds[i].func==NULL)
            myPrintk(0x7, "\nUnknown command:%s\n",args[0]);

    }
}
```

目录组织



Makefile组织

```
SRC_RT=/home/lps3025/workspace/lab3/
#SRC_RT=$(shell pwd)

CROSS_COMPILE=
ASM_FLAGS= -m32 --pipe -Wall -fasm -g -O1 -fno-stack-protector
C_FLAGS =  -m32 -fno-stack-protector -g

.PHONY: all
all: output/myOS.elf

MULTI_BOOT_HEADER=output/multibootheader/multibootHeader.o
include $(SRC_RT)/myOS/Makefile
include $(SRC_RT)/userApp/Makefile

OS_OBJS       = ${MYOS_OBJS} ${USER_APP_OBJS}

output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
    ${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS} -o output/myOS.elf

```

```
output/%.o : %.S
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${ASM_FLAGS} -c -o $@ $<

output/%.o : %.c
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${C_FLAGS} -c -o $@ $<

clean:
    rm -rf output
```

基本规则：

```
output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
    ${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS} -o output/myOS.elf
```

将各级子目录下makefile文件中定义好的.o文件作为依赖文件，按照myOS.ld规则生成目标文件myOS.elf，放在output文件夹下。

代码布局说明

链接描述文件myOS.ld如下：

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(start)

SECTIONS {
    . = 1M;
    .text : {
        *(.multiboot_header)
        . = ALIGN(8);
        *(.text)
    }

    . = ALIGN(16);
    .data       : { *(.data*) }

    . = ALIGN(16);
    .bss        :
    {
        __bss_start = .;
        _bss_start = .;
        *(.bss)
        __bss_end = .;
    }
    . = ALIGN(16);
```

```
    _end = .;
    . = ALIGN(512);
}
```

说明： 首先定位到1M地址处。可执行文件的.text段从此处开始。开始存放输入文件(MultibootHeader.S)的.multiboot_header段[12字节]，往后对齐8字节后，再存放所有输入文件的.text段。 往后对齐16字节后，接着存放可执行文件的.data段，包含的内容即为所有输入文件的.data段。往后对齐16字节后，接着存放可执行文件的.bss段，包括所有输入文件中一些未初始化的变量。bss段结束后依次向后对齐16字节和512字节。

编译过程说明

直接在终端运行./source2run.sh即可。具体过程是先make编译，编译成功后再执行命令

```
qemu-system-i386 -kernel output/myOS.elf -serial pty &
```

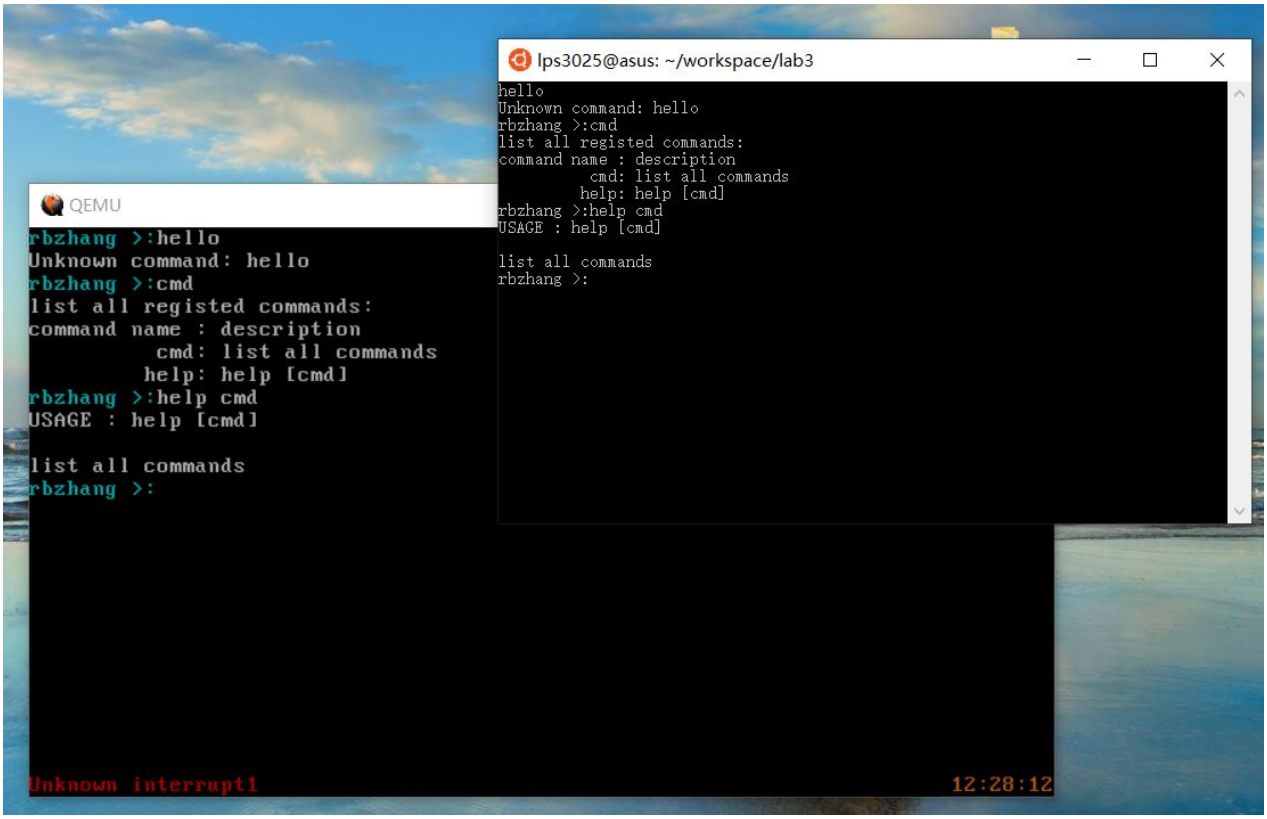
将串口重定向到伪终端，运行时会告知具体是哪个，再据此输入

```
sudo screen /dev/pts/1   #假设是/dev/pts/1
```

通过伪终端输入命令。

运行和运行结果说明

运行结果如下：



说明：分别在命令行输入hello，cmd，help cmd。结果分别报错、列出所有内置命令、列出cmd帮助信息。左下角显示中断信息，右下角显示墙钟。

遇到的问题和解决方案说明

实现startShell()时，为了分析读入的一行输入，需要按空格对输入的字符串进行分割。为此，在string.c中添加了一个strtok()专门做字符串分割。shell的具体实现参考了这篇博文:[译] 教程 - 用 C 写一个 Shellhttps://nettee.github.io/posts/2019/Tutorial-Write-a-Shell-in-C/。

发现滚屏功能会导致VGA上左下角的中断信息和右下角的时钟往屏幕上方移动。于是修改了滚屏的范围，使滚屏只对屏幕前24行生效，第25行只用来显示中断和时钟。